Solving Very Large Scale Linear SVM Using Multiple Processors

Prateek Jindal, Dan Roth, L.V. Kale

{jindal2,danr,kale}@illinois.edu

Dept. of Computer Science, UIUC

Abstract

SVMs have been used for long for data classification. While solving very large problems, one may encounter hundreds of thousands of features and a large number of training vectors. A natural solution to solving problems with large datasets is to use multiple processors. In this paper, we discuss the parallel implementation of Liblinear (Hsieh et. al. (2008)) which is a very good sequential tool for using SVMs.

Introduction

Support vector machines (SVM) are useful for data classification. Given a set of instance-label pairs (x_i, y_i), i = 1, …, l, $x_i \in R^n$; $y_i \in \{-1, +1\}$, SVM requires the solution of the following unconstrained optimization problem:

$$\min_w \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi(w; x_i, y_i)$$

where $\xi(w; x_i, y_i)$ is a loss function, and C > 0 is a penalty parameter. Two common loss functions are:

$$max(1 - y_i w^T x_i, 0) and \max(1 - y_i w^T x_i, 0)^2$$

The former is called L1-SVM, while the latter is L2-SVM. In some applications, an SVM problem appears with a bias term b. One often deal with this term by appending each instance with an additional dimension:

$$x_i^T \leftarrow [x_i^T, 1] \qquad\qquad w^T \leftarrow [w^T, b]$$

The above problem is often referred to as the primal form of SVM. One may instead solve its dual problem:

$$\min_\alpha f(\alpha) = \frac{1}{2} \alpha^T \overline{Q} \alpha - e^T \alpha$$

$$subject\ to \quad 0 \leq \alpha_i \leq U, \forall i$$

where $\overline{Q} = Q + D$, D is a diagonal matrix, and $Q_{ij} = y_i y_j x_i^T x_j$. For L1-SVM, U=C and $D_{ii} = 0, \forall i$. For L2-SVM, $U = \infty \; and \; D_{ii} = \frac{1}{2C}, \forall i$.

Existing Solutions

Recently, many methods have been proposed for linear SVM in large-scale scenarios. For L1-SVM, Zhang (2004), Shalev-Shwartz et al. (2007), Bottou (2007) propose various stochastic gradient descent methods. Collins et al. (2008) apply an exponentiated gradient method. SVM^{perf} (Joachims, 2006) uses a cutting plane technique. Joachims (2006), Smola et al. (2008) and Collins et al. (2008) solve SVM via the dual. Others consider the primal form. The decision of using primal or dual is of course related to the algorithm design. Very recently, Chang et al. (2008) propose using coordinate descent methods for solving primal L2-SVM

Hsieh et. al. (2008) propose a dual coordinate descent method for linear SVM. Their implementation is known by the name Liblinear. They show that their method is much faster than state of the art solvers such as Pegasos and SVM^{perf}. And it reaches an ϵ accurate solution in $O\left(\log\left(\frac{1}{\epsilon}\right)\right)$ iterations. We use Liblinear as a base tool for developing the parallel implementation to solve very large scale SVM.

Working of Liblinear

.

Algorithm 1: A dual coordinate descent method for Linear SVM

- Given α and the corresponding $w = \sum_i y_i \alpha_i x_i$.
- While α is not optimal
  - For i = 1, 2, … , l
    - $G = y_i w^T x_i - 1 + D_{ii}\alpha_i$
    - $PG = \begin{cases} \min(G, 0), & if \; \alpha_i = 0, \\ \max(G, 0), & if \; \alpha_i = U, \\ G, & if \; 0 < \alpha_i < U \end{cases}$
    - if |PG| ≠ 0,
      - $\overline{\alpha_i} \leftarrow \alpha_i$
      - $\alpha_i \leftarrow \min(\max(\alpha_i - G/\overline{Q_{ii}}, 0), U)$
      - $w \leftarrow w + (\alpha_i - \overline{\alpha_i}) y_i x_i$

Coordinate descent method for L1- and L2-SVM (Hsieh et. al. (2008)) is described in Algorithm 1. The optimization process starts from an initial point $\alpha^0 \in R^l$ and generates a sequence of vectors $\{\alpha^k\}_{k=0}^{\infty}$. Hsieh et. al. (2008) refer to the process from α^k to α^{k+1} as an outer iteration. In each outer iteration, there are l inner iterations, so that sequentially $\alpha_1, \alpha_2, \ldots, \alpha_l$ are updated. Each outer iteration thus generates vectors $\alpha^{k,i} \in R^l, i = 1, \ldots, l+1, such \; that \; \alpha^{k,1} = \alpha^k, \alpha^{k,l+1} = \alpha^{k+1}$, and

$$\alpha^{k,i} = [\alpha_1^{k+1}, \dots, \alpha_{i-1}^{k+1}, \alpha_i^k, \dots, \alpha_l^k]^T, \quad \forall i = 2, \dots, l.$$

Limitations of Liblinear

Although Liblinear is able to efficiently learn the linear classifiers, it doesn't scale well for very large problem sizes. Liblinear stores all the training vectors into the memory as <Feature, Value> pairs. As a result, Liblinear requires the memory space which increases linearly with the size of the dataset. Figure 1 shows the memory consumed by Liblinear as a function of the training data size. The total number of features used in this experiment was 100,000. As we increase the number of training vectors from 2000 to 20,000, the memory requirements increase from 0.74 GB to 7.4 GB. So, it is clear that we can't use Liblinear for training very large datasets.

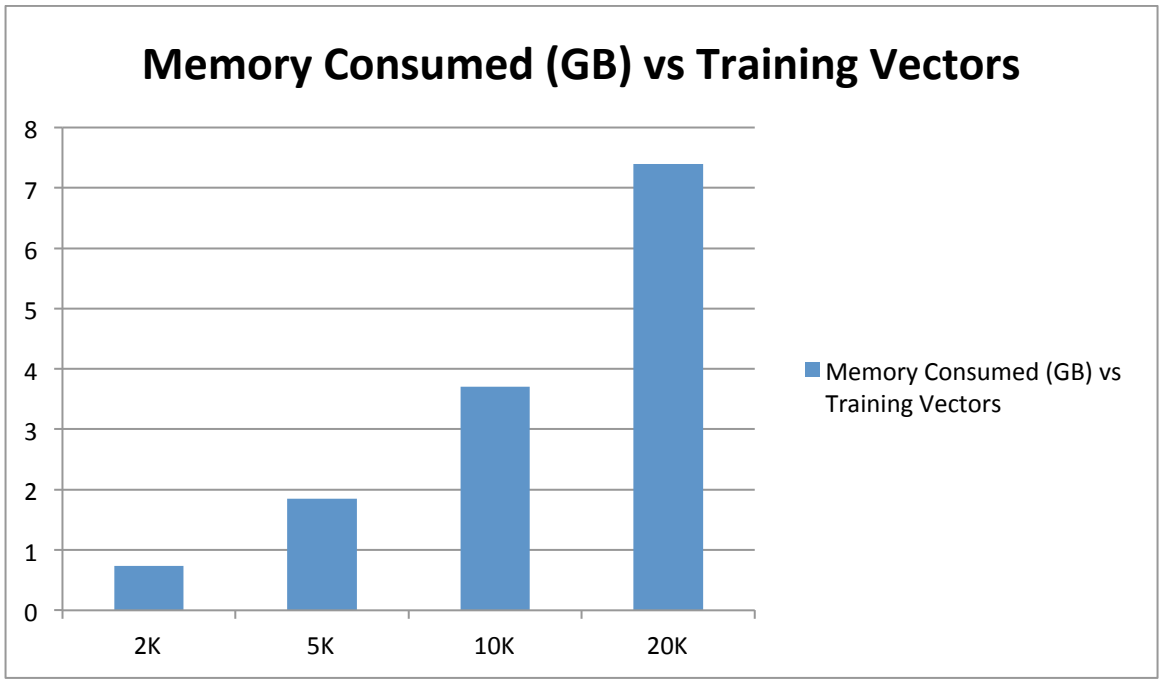

Figure 1: A figure showing the memory consumed by Liblinear as the number of training vectors are increased

Scaling Liblinear to very large datasets

One solution to overcome the problem of insufficient memory is to use the disk space. We note that in each inner iteration, Liblinear needs the weight vector and 1 training vector. Thus, we can swap the training vectors into and out of memory alternately and thus use only a constant amount of memory. But the problem here is that the Liblinear needs all the training vectors for every outer iteration. This would lead to a lot of swaps into and out of memory and thus would be very inefficient. An alternative solution is to use multiple processors in parallel.

We have implemented 2 solutions to using multiple processers. Each of the solutions is described below.

Solution 1

In the first solution, we implemented the exact version of Liblinear in parallel. Figure 2 shows the schematic view of parallel liblinear. This figure shows that we maintain a set of processes. Each of the processes is responsible for a subset of the data. In each outer iteration of Liblinear, we see that the entire training set is scanned once. The training vectors are considered one by one in the inner iterations. Each inner iteration updates the weight vector. The next iteration uses the modified weight vector. This makes the exact implementation inherently sequential.

The master process initializes the weight vector to zero. It sends the weight vector to the first process. This process runs one outer iteration on its part of the training set. The modified weight vector is passed on to 2nd process. In this way, the weight vector is passed along the chain of processes. Along with the weight vector, we also pass the maximum and the minimum projected gradient that has been obtained so far. The last process in the chain passes the weight vector back to the master process. The master process keeps track of the total number of iterations that have executed so far. From the maximum and minimum projected gradient, the master process determines whether the required convergence has been reached or not according to the following relation:

$$if\ \left((PGmax - PGmin) \leq eps\right) \quad then\ \ stop.$$

where eps is provided by the user. A value of 0.1 is good for most purposes. The problem with the exact implementation is that only one of the processes is active at any moment. But it solves the problem of insufficient memory by distributing the training set among different processors.

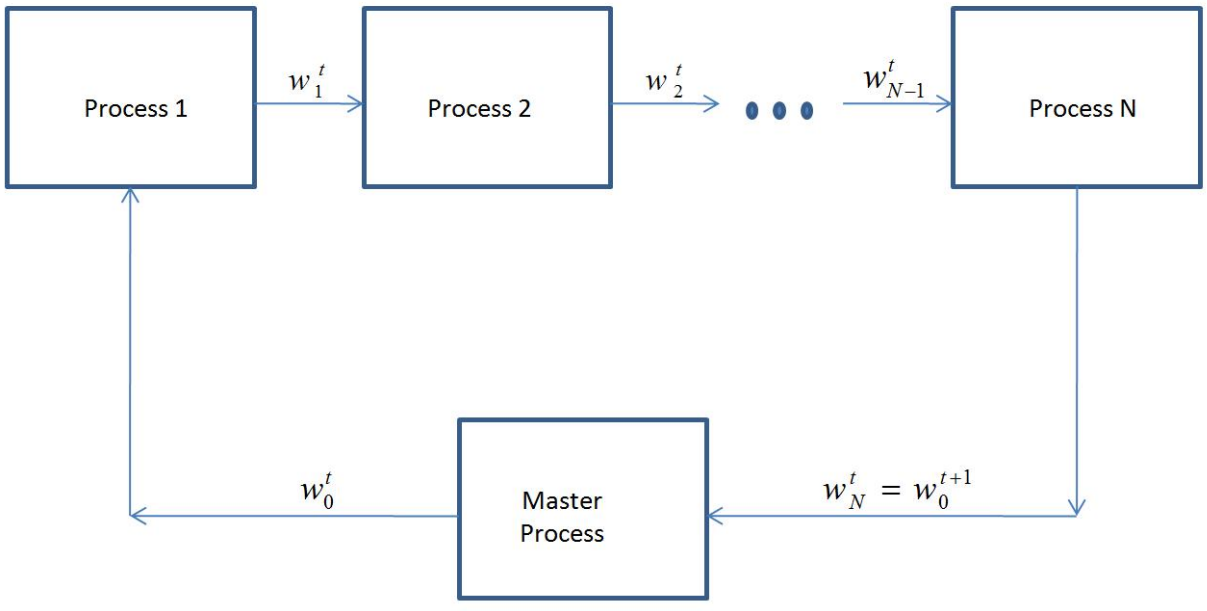

Figure 2: Parallel Implementation of Liblinear (Solution 1)

Solution 2

To improve the resource utilization of parallel Liblinear, we modified the above version by training the different parts of the dataset concurrently. Figure 3 shows the schematic view of this implementation. In this implementation, we divide the training set among different processes as in the previous solution. But, the master process doesn't send the weight vector to process 1 as in the previous solution. Instead, the master process broadcasts the weight vector to all the processes. Now, each of the processes carries out one outer iteration on its part of the training set concurrently. After one outer iteration, all the processes send their weight vector to the master process. The master process computes the new weight vector from the received weight vectors according to the following equation:

$$w^{t+1} = \sum_i (w_i^{t+1})/N$$

where w_i^{t+1} is the weight vector output by i^{th} process at the end of t^{th} iteration. And w^{t+1} is the weight vector to be broadcasted to all the processes at the beginning of $(t+1)^{th}$ iteration. As in the previous solution, all the sub-processes also send their maximum and minimum projected gradient to the master process. The master process determines whether the convergence has been reached or not by using maximum and minimum projected gradients as in the previous solution. If the convergence has not been reached yet, the weight vector is broadcasted again and the process continues until convergence.

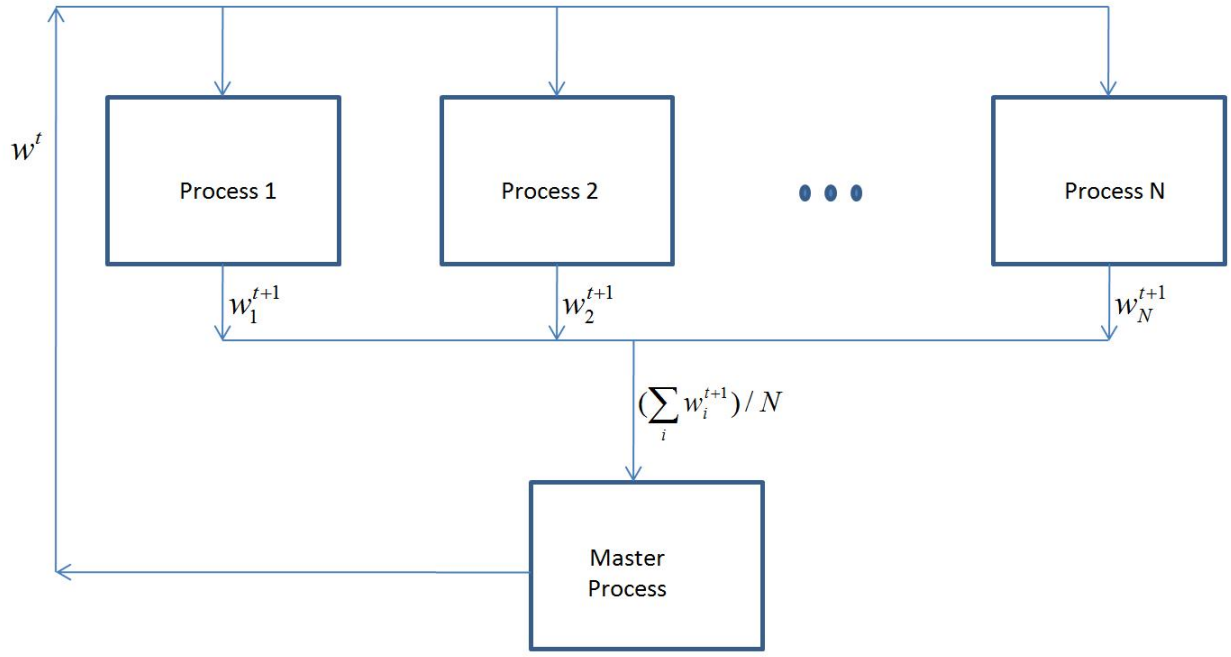

Figure 3: Parallel Implementation of Liblinear (Solution 2)

Using Parallel Implementations inside LBJ

The parallel implementations have been developed using the Charm++ Runtime System. Calling the parallel implementation from LBJ (Rizzolo and Roth (2007)) is very simple. The name of the binary which implements the parallel version is pliblinear. Once the Charm++ has been installed, one needs to make the following system call from inside LBJ to use the parallel implementation.

./charmrun +pn ./pliblinear <path of the training file>

where n is the number of processors to be used.

Results

Next, we compare the convergence behavior of the 2 parallel solutions to Liblinear. The machine used for these experiments had the following characteristics:

CPU: Intel Xeon 2.00 GHz 8-core
Memory: 5 GB
Processors Used for Experiments: 4

Table 1 shows the characteristics of the datasets used in the experiments. We trained both versions of parallel Liblinear on these datasets. Figures 4 and 5 below show the convergence behavior of the 2 solutions. The steeper descent indicates the faster convergence. We find that solution 1 converges faster for 1st dataset. For 2nd dataset, both the solutions converge almost equally fast. These figures reveal that both the solutions have good convergence properties. Table 2 shows the time taken by 2 parallel implementations. We find that 2nd implementation is about 2.6 times faster than the first.

	Features	Training Vectors	Non-zero Elements
Dataset 1	10,000	4,000	20,000,000
Dataset 2	2,000	8,000	8,000,000

Table 1: Characteristics of the datasets used for experiments

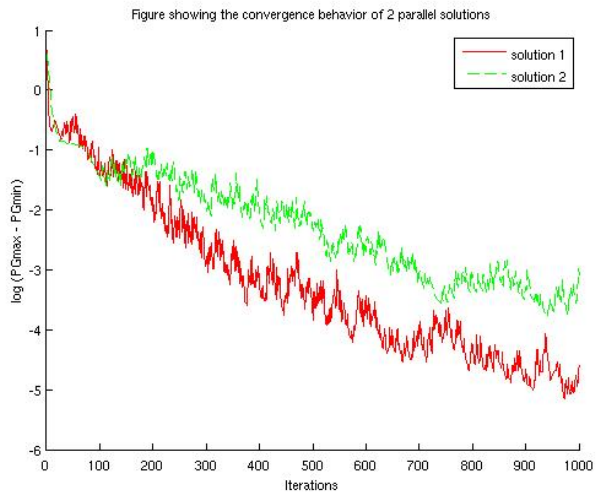

Figure 4: Convergence behavior for Dataset 1

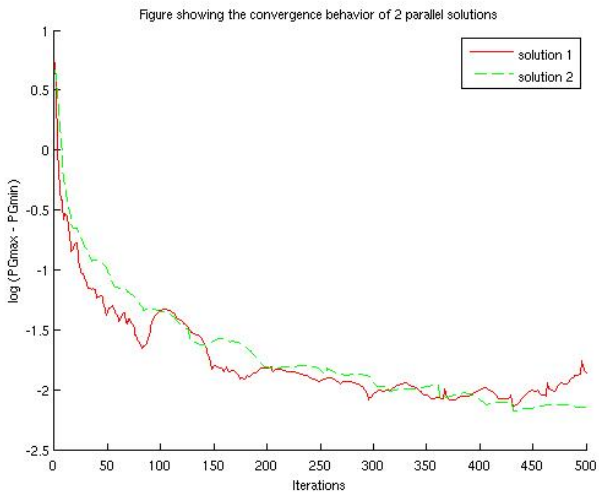

Figure 5: Convergence behavior for Dataset 2

	Solution 1	Solution 2
Dataset 1	196	75
Dataset 2	39	15

Table 2: Comparison of time taken (in seconds) by two parallel implementations

Conclusion

First of all, we described the limitations of Liblinear when it is used to train very large datasets. Then we discussed the ways to overcome this limitation by parallelizing the Liblinear. The first solution that we described used multiple processors sequentially and hence was less efficient. To overcome this limitation, we implemented a second version of parallel Liblinear. The results show that the 2nd solution also converges very well.

References

Bottou, L. (2007). Stochastic gradient descent examples. http://leon.bottou.org/projects/sgd.

Chang, K.-W., Hsieh, C.-J., & Lin, C.-J. (2008). Coordinate descent method for large-scale L2-loss linear SVM. Journal of Machine Learning Research, 9, 1369-1398.

Collins, M., Globerson, A., Koo, T., Carreras, X., & Bartlett, P. (2008). Exponentiated gradient algorithms for conditional random fields and max-margin Markov networks. JMLR.

Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and Sellamanickam Sundararajan. A dual coordinate descent method for large-scale linear SVM. In Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML), 2008.

Joachims, T. (2006). Training linear SVMs in linear time. ACM KDD.

Rizzolo, N., and Roth, D. 2007. Modeling Discriminative Global Inference. In Proceedings of the First International Conference on Semantic Computing (ICSC), 597–604. Irvine, California: IEEE.

Shalev-Shwartz, S., Singer, Y., & Srebro, N. (2007). Pegasos: primal estimated sub-gradient solver for SVM. ICML.

Smola, A. J., Vishwanathan, S. V. N., & Le, Q. (2008). Bundle methods for machine learning. NIPS.

Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. ICML.